DEVELOPMENT OF PLATFORM INDEPENDENT INSTANT MESSENGER

BY

MAHANTH KUMAR BEERAKA-031005

ASHISH GUPTA-031022

VIVEK AGARWAL-031003

विद्या तत्व ज्योतिसम:

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

MAY-2007

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
(Established by H.P. State Legislative vide Act No. 14 of 2002)
Waknaghat, P.O. Dumehar Bani, Kandaghat, Distt. Solan - 173215 (H.P.) INDIA
Website : www.juit.ac.in
Phone No. (91) 01792-257999 (30 Lines).
Fax: (91) 01792 245362

Dr. Sarit Pal
Assistant Professor
Department of Electronics and Communication Engineering

CERTIFICATE

This is to certify that the work entitled, "DEVELOPMENT OF PLATFORM INDEPENDENT INSTANT MESSENGER" submitted by Mahanth Kumar Beeraka, Ashish Gupta and Vivek Agarwal in partial fulfillment for the award of degree of Bachelor of Technology in Electronics and communication engineering of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Sarit Pal
(Sarit Pal)
Supervisor

ACKNOWLEDGEMENT

We wish to express our earnest gratitude to **Dr. Sarit Pal**, for providing us invaluable guidance and timely suggestions throughout the length of the project. We'd also like to thank him for his moral support in times when the project was losing pace.

We would also like to thank **Dr. Sunil Bhooshan**, HOD Electronics and Communication Department of Jaypee University of Information Technology, Waknaghat for his support and wise suggestions.

We would like to the thank the faculty of the Electronics and Communication Department of Jaypee University of Information Technology, Waknaghat for their valuable suggestions that made helped us improve our project.

Mahanth Kumar Beeraka Ashish Gupta Vivek Agarwal

ABSTRACT

Study of Java Programming and Network Programming through Java has been done. Report of this study is presented. A Centralized Instant Messenger application using the same was developed, the details of which are presented.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 2 Programming language (Java)

Chapter 3 The Project

CHAPTER 1
INSTANT MESSAGING

Introduction

Instant Messaging (IM) is a fast and easy method of communicating to another individual while online in a real time, real time is defined as the immediate response by a computer system. One of the biggest advantages of IM programs is that they allow users the ability to chat with someone as if you were sitting right next to them, having an actual conversation, but instead speaking you are typing.

To communicate with friends, family members and/or co workers, you need to your friends, family members and/or co workers user names. The user names can be stored in a program and can be referred to as contact list. Lastly, both parties need to have the same instant messenger program, such as AOL and MSN, to communicate.

Unfortunately, none of the programs mentioned above are compatible with each other. There are, however, a few programs available that integrate the different IM programs into one interface.

Besides being able to see if one of your buddy/contacts is online, Instant messenger programs also allow users to send each other text message, photos, pictures and word document files in real time. In addition, some IM programs have voice and Web cam features to enhance one's experience. Most IM programs provide ways for creating your own chat room for multiple users.

Popular instant messaging services on the public Internet include .NET Messenger Service, AOL Instant Messenger, Excite/Pal, Gadu-Gadu, Google Talk, iChat, ICQ, Jabber, Qnext, QQ, Skype and Yahoo! Messenger. These services owe many ideas to an older (and still popular) online chat medium known as Internet Relay Chat (IRC).

1.1 BENEFITS

Instant messaging typically boosts communication and allows easy collaboration. In contrast to e-mails, the parties know whether the peer is available. Most systems allow the user to set an online status or away message so peers get notified whenever the user is available, busy, or away from the computer. On the other hand, people are not forced to reply immediately to incoming messages. This way, communication via instant messaging can be less intrusive than communication via phone, which is partly a reason why instant messaging is becoming more and more important in corporate environments.

1.2 EARLY DEVELOMENTS

An early and partial form of messaging systems was implemented on private computer networks such as the PLATO system in the early 1970s. It was also available in the 1970s as the "talk" program. Later the Unix/Linux "talk" messaging systems were widely used by engineers and academics in the 1980s and 1990s to communicate across the internet. On single line bulletin board systems (BBS), the system operator (sysop) and the single caller online could typically chat with one another. One's typing appeared in real time for the other person as an instant message equivalent.

Recently, many instant messaging services have begun to offer video conferencing features, Voice Over IP (VoIP) and web conferencing services. Web conferencing services integrate both video conferencing and instant messaging capabilities. Some newer instant messaging companies are offering desktop sharing, IP radio and IPTV to the voice and video features.

What really characterizes instant messaging from other forms of text messaging applications is the use of "presence" which enables the user of an instant messaging application to rendezvous with his/her counterparties and see their status of availability.

1.2.1 What has been done?

Messengers can be characterized as -

- Centralized or decentralized.
- Platform independent or dependent.
- Various Protocols based.
- Different Programming languages.

1.2.2 What are the various properties that are available in present messengers?

- Instant messaging
- Sharing files and folders
- PC-to-phone calls
- Games and applications

Many messengers have been made and presently being widely used on various networks (LAN) which connects various computers as clients in the network. These messengers have various properties (mentioned above).

1.2.3 Some messengers in use with their properties are -

- Windows Live Messenger.
- Outlook Messenger.
- IP Messenger.
- GOIM

1.2.3(1) ***Windows Live Messenger (WLM)***, formerly and still commonly referred to as MSN Messenger or MSN, is a freeware instant messaging client for Windows XP, Windows Vista, and Windows Mobile, first released on July 22, 1999 by Microsoft. "MSN Messenger" is often also used to refer to the .NET Messenger Service (the protocols and servers that allow the system to operate) rather than any particular client. Most major multi-protocol clients can also connect to the service.

Windows Live Messenger uses the Mobile Status Notification Protocol (MSNP) over TCP (and optionally over HTTP to deal with proxies) to connect to the .NET Messenger Service.Windows Live messenger provides various services which include sharing files and folders, pc to phone calls, instant messaging.

1.2.3(2) ***Outlook Messenger*** is a Concept LAN chat for interactive communication within an Office network (LAN). In addition to the usual rich text chat, voice chat, group chat, send file, reminders and alert note functions, Outlook Messenger can be plugged into MS Outlook, allowing the users to share Outlook e-mails, contacts, and appointments. The Value added feature 'Remote Desktop Sharing' lets users access and control a remote computer.

This LAN Instant Messaging does not require any internet connection, and it works across Ethernet port using TCP/IP protocol. As this module is built with peer to peer architecture, it requires no Server and IP configuration. Installing LAN Messaging tool is very easy, and system administrator help is not necessary. Just install the software in all the computers, and the program is ready to use. You need not add User List manually, its automatically done.

1.2.3(3) ***IP Messenger*** is pop style LAN messaging software for multiple platforms. It is based on TCP and UDP/IP Protocols. This software does not require server machine means that it is not centralized, which uses server-client based approach, rather it is decentralized. The user can connect to this messenger independent of any server.

1.2.3(4) ***Gamers Own Instant Messenger (GOIM)*** is a full featured open source Instant Messaging client based on the open source Jabber/XMPP protocol. Jabber has many advantages over most other IM protocols - It provides an open protocol which is implemented by dozens of servers and clients. It has a decentralized server structure (like email) so you have the freedom of choice which server you are using as well as which client you want to use. And if you are unhappy with your client just switch to another one - your contact list (roster) will go with you.

It was primarily developed to appeal to gamers so it not only transmits your presence status (like available, away, do not disturb) but also on which server you are currently

playing which game. This way you can see where your friends are playing and join them with a single click.

GOIM is based on the java programming. So all of GOIM's functionality is cross platform and can be used on Microsoft Windows, Linux and Mac OS - The only exception being platform specific code to detect game connections as well as the InGameMessenger.

1.3 METHODOLOGY

1.3.1 What are the various approaches that can be adopted?

- Client/Server approach.
- Peer to Peer approach.

1.3.1(1) ***Client/server*** is network architecture which separates the client (often an application that uses a graphical user interface) from the server. Each instance of the client software can send requests to a server or application server. There are many different types of servers; some examples include: a file server, terminal server, or mail server. While their purpose varies somewhat, the basic architecture remains the same.

a) Characteristics of a server:

- Passive (slave)
- Waits for requests
- Upon receipt of requests, processes them and then serves replies

b) Characteristics of a client:

- Active (master)
- Sends requests
- Waits for and receives server replies

Servers can be stateless or stateful. A stateless server does not keep any information between requests. A stateful server can remember information between requests. The

scope of this information can be global or session. A HTTP server for static HTML pages is an example of a stateless server while Apache Tomcat is an example of a stateful server.

A **client** is a computer system that accesses a (remote) service on another computer by some kind of network. The term was first applied to devices that were not capable of running their own stand-alone programs, but could interact with remote computers via a network. These dumb terminals were clients of the time-sharing mainframe computer.

c) Types of client

Clients are generally classified as:

	Local storage	Local processing
Thin Client	No	No
Hybrid Client	No	Yes
Fat Client	Yes	Yes

A **fat client** (also known as a thick client or rich client) is a client that performs the bulk of any data processing operations itself, but does not necessarily rely on the server. The fat client is most common in the form of a personal computer, as the PC or laptops can operate independently.

A **thin client** is a minimal sort of client. Thin clients use the resources of the host computer. A thin client's job is generally just to graphically display pictures provided by an application server, which performs the bulk of any required data processing.

A **hybrid client** is a mixture of the above. Similar to fat client, it is processing locally, but rely on the server for the storage. This relatively new approach offers features from both the fat client (multimedia support, high performance) and the thin client (high manageability, flexibility).

1.3.1(2) ***Peer-to-peer (or P2P)*** computer network refers to any network that does not have fixed clients and servers, but a number of peer nodes that function as both clients and servers to the other nodes on the network. This model of network arrangement is contrasted with the client-server model. Any node is able to initiate or complete any supported transaction. Peer nodes may differ in local configuration, processing speed, network bandwidth, and storage quantity.

Technically, a true peer-to-peer application must implement only peering protocols that do not recognize the concepts of "server" and "client". Such pure peer applications and networks are rare. Most networks and applications described as peer-to-peer actually contain or rely on some non-peer elements, such as DNS. Also, real world applications often use multiple protocols and act as client, server, and peer simultaneously, or over time.

CHAPTER 2

PROGRAMMING LANGUAGE (JAVA)

Introduction

In discussing Java, it is important to distinguish between the Java programming language, the Java Virtual Machine, and the Java platform. The Java programming language is the language in which Java applications (including applets, servlets, and JavaBeans components) are written. When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces).

2.1 The Java Programming Language

The Java programming language is a state-of-the-art, object-oriented language that has syntax similar to that of C. By keeping the language simple, the designers also made it easier for programmers to write robust, bug-free code.

2.2 The Java Virtual Machine

The Java Virtual Machine, or Java interpreter, is the crucial piece of every Java installation. By design, Java programs are portable, but they are only portable to platforms to which a Java interpreter has been ported. Sun ships VM implementations for its own Solaris operating system and for Microsoft Windows and Linux platforms.

Many other vendors, including Apple and various commercial UNIX vendors, provide Java interpreters for their platforms.

Although interpreters are not typically considered high-performance systems, Java VM performance is remarkably good and has been improving steadily. Of particular note is a VM technology called just-in-time (JIT) compilation, whereby Java byte codes are converted on-the-fly into native-platform machine language, boosting execution speed for code that is run repeatedly.

2.3 The Java Platform

The Java platform is just as important as the Java programming language and the Java Virtual Machine. All programs written in the Java language rely on the set of predefined classes that comprise the Java platform. Java classes are organized into related groups known as ***packages***. The Java platform defines packages for functionality such as input/output, networking, graphics, user-inter face creation, security, and much more.

The Java 1.2 release was a major milestone for the Java platform. This release almost tripled the number of classes in the platform and introduced significant new functionality. In recognition of this, Sun named the new version the Java 2 Platform.

It is important to understand what is meant by the term platform. To a computer programmer, a platform is defined by the APIs he or she can rely on when writing programs. These APIs are usually defined by the operating system of the target computer. Thus, a programmer writing a program to run under Microsoft Windows must use a different set of APIs than a programmer writing the same program for the Macintosh or for a Unix-based system. In this respect, Windows, Macintosh, and Unix are three distinct platforms.

Java is not an operating system. Nevertheless, the Java platform provides APIs with a comparable breadth and depth to those defined by an operating system. With the Java platform, you can write applications in Java without sacrificing the advanced features available to programmers writing native applications targeted at a particular

underlying operating system. An application written on the Java plat-for m runs on any operating system that supports the Java platform. This means you do not have to create distinct Windows, Macintosh, and UNIX versions of your programs, for example. A single Java program runs on all these operating systems, which explains why "Write once, run anywhere" is Sun's motto for Java.

2.4 Key Benefits of Java

This section explores some of the key benefits of Java.

2.4.1 Write Once, Run Anywhere

Sun identifies "Write once, run anywhere" as the core value proposition of the Java platform. Translated from business jargon, this means that the most important promise of Java technology is that you only have to write your application once—for the Java platform—and then you'll be able to run it ***anywhere.***

Anywhere, that is, that supports the Java platform. Fortunately, Java support is becoming ubiquitous. It is integrated, or being integrated, into practically all major operating systems. It is built into the popular web browsers, which places it on virtually every Internet-connected PC in the world.

2.4.2 Security

Another key benefit of Java is its security features. The Java platform allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: untrusted code cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.

2.4.3 Network-Centric Programming

Sun's corporate motto has always been "The network is the computer." The designers of the Java platform believed in the importance of networking and

designed the Java platform to be network-centric. From a programmer's point of view, Java makes it easy to work with resources across a network and to create network-based applications using client/server or multitier architectures.

2.4.4 Dynamic, Extensible Programs

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called ***classes***. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

2.4.5 Internationalization

When it was created, Java was the only commonly used programming language that had internationalization features at its core. While most programming languages use 8-bit characters that represent only the alphabets of English and Western European languages, Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world. Java's internationalization features are not restricted to just low-level character representation, however. The features permeate the Java platform, making it easier to write internationalized programs with Java than it is with any other environment.

2.4.6 Performance

As described earlier, Java programs are compiled to a portable intermediate form known as byte codes, rather than to native machine-language instructions. The Java Virtual Machine runs a Java program by interpreting these portable byte-code instructions. This architecture means that Java programs are faster than programs or scripts written in purely interpreted languages, but they are typically slower than C and C++ programs compiled to native machine language. Keep in mind, however, that although Java programs are compiled to byte code, not all of the Java platform is implemented with interpreted byte codes. For efficiency, computationally intensive

portions of the Java platform—such as the string-manipulation methods—are implemented using native machine code.

Java is a portable, interpreted language; Java programs run almost as fast as native, non-portable C and C++ programs. Performance used to be an issue that made some programmers avoid using Java. Now, with the improvements made in Java 1.2, 1.3, 1.4 and 1.5 performance issues should no longer keep anyone away.

2.4.7 Programmer Efficiency and Time-to-Market

The final and perhaps most important, reason to use Java is that programmers like it. Java is an elegant language combined with a powerful and (usually) well-designed set of APIs. Programmers enjoy programming in Java and are often amazed at how quickly they can get results with it. Because Java is a simple and elegant language with a well-designed, intuitive set of APIs, programmers write better code with fewer bugs than for other platforms, again reducing development time.

2.5 Understanding Streams

A **stream** is an abstract representation of an input or output device that is a source of, or destination for, data. You can write data to a stream and read data from a stream. Stream as a sequence of bytes that flows into or out of a program can be visualized.

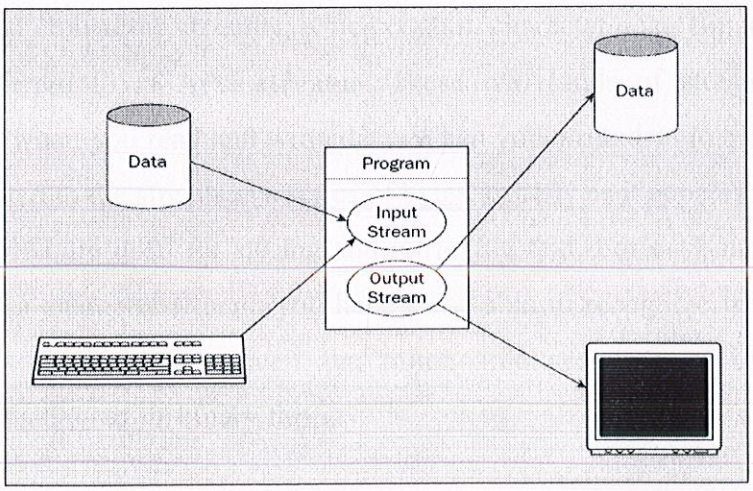

Fig.2-1 Illustrates how physical devices map to streams.

2.5.1 Input and Output Streams

When you write data to a stream, the stream is called an **output stream**. The output stream can go to any device to which a sequence of bytes can be transferred, such as a file on a hard disk, or a phone line connecting your system to a remote system. An output stream can also go to your display screen When you write to your display screen using a stream, it can display characters only, not graphical output.You read data from an **input stream**. In principle, this can be any source of serial data, but is typically a disk file, the keyboard, or a remote computer.

The main reason for using a stream as the basis for input and output operations is to make your program code for these operations independent of the device involved. This has two advantages. First, you don't have to worry about the detailed mechanics of each device, which are taken care of behind the scenes. Second, your program will work for a variety of input/output devices without any changes to the code. Stream input and output methods generally permit very small amounts of data, such as a single character for byte, to be written or read in a single operation. Transferring data to or from a stream like this may be extremely inefficient, so a stream is often equipped with a **buffer** in memory, in which case it is called a **buffered stream**.

2.5.2 Binary and Character Streams

The java.io package supports two types of streams—**binary streams,** which contain binary data, and **character streams,** which contain character data. Binary streams are sometimes referred to as **byte streams**. These two kinds of streams behave in different ways when you read and write data. When you write data to a binary stream, the data is written to the stream as a series of bytes, exactly as it appears in memory. Character streams are used for storing and retrieving text. For each numerical value you read from a character stream, you have to be able to recognize where the value begins and ends and then convert the **token**—the sequence of characters that represents the value—to its binary form.

When you write strings to a stream as character data, by default the Unicode characters are automatically converted to the local representation of the characters in

the host machine, and these are then written to the stream. When you read a string, the default mechanism converts the data from the stream back to Unicode characters from the local machine representation.

2.5.3 The Classes for Input and Output

The package java.io contains the classes that provide the foundation for Java's support for stream I/O:

Class	Description
InputStream	The base class for byte stream input operations.
OutputStream	The base class for byte stream output operations.

InputStream and OutputStream are both **abstract** classes. As you know, you cannot create instances of an abstract class—these classes serve only as a base from which to derive classes with more concrete input or output capabilities. Generally, the InputStream and OutputStream classes, and their subclasses, represent byte streams and provide the means of reading and writing binary data as a series of bytes.

2.5.4 Basic Input Stream Operations

The InputStream class includes three methods for reading data from a stream: read( ), read(byte[ ] array), read(byte[ ] array, int offset, int length).

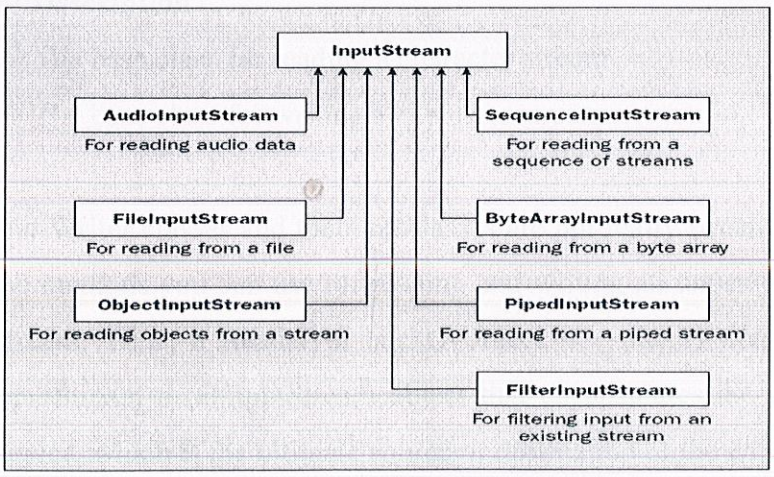

Fig.2-2 Subclasses of InputStream

2.5.5 Basic Output Stream Operations

The OutputStream class contains three write( ) methods for writing binary data to the stream. As can be expected, these mirror the read( ) methods of the InputStream class. This class is also abstract, so only subclasses can be instantiated.

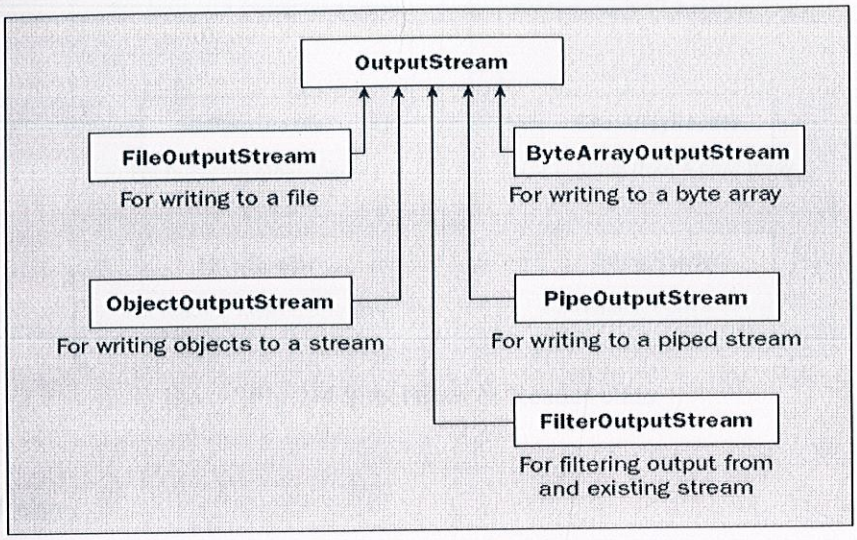

Fig 2-3 Subclasses of OutputStream

2.5.6 Stream Readers and Writers

Stream readers and **writers** are objects that can read and write byte streams as character streams. So a character stream is essentially a byte stream fronted by a reader or a writer. The base classes for stream readers and writers are:

Class	Description
Reader	The base class for reading a character stream
Writer	The base class for writing a character stream

The Reader and Writer classes and their subclasses are not really streams themselves, but provide the methods you can use for reading and writing an underlying stream as a character stream. Thus, a Reader or Writer object is typically created using an underlying InputStream or OutputStream object that encapsulates the connection to the external device, which is the ultimate source or destination of the data.

2.5.6(1) *Readers*

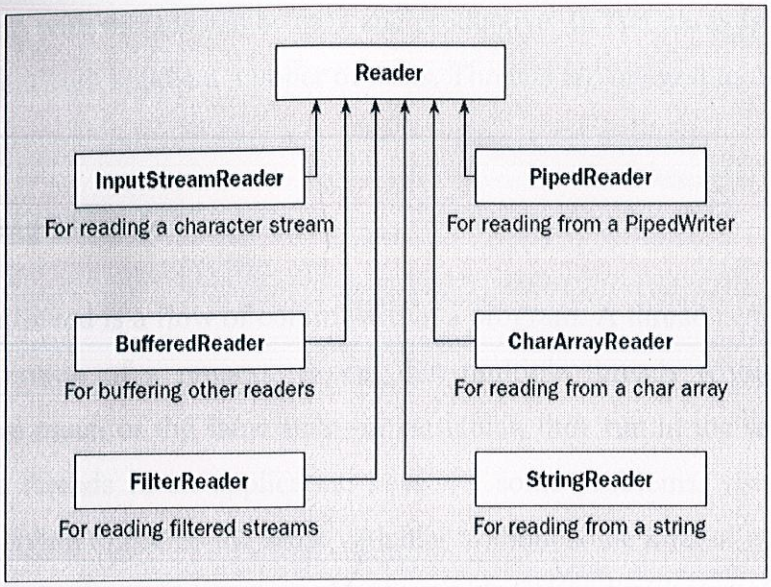

Fig 2-4 Subclasses of Reader class

2.5.6(2) *Writers*

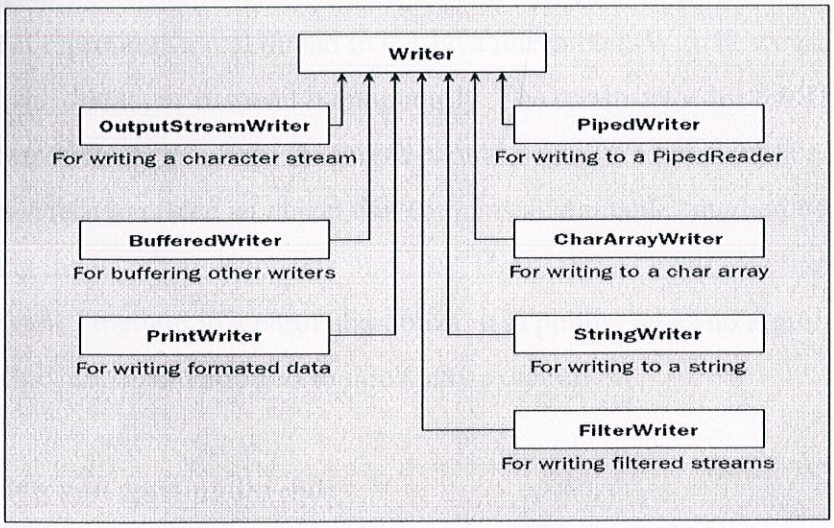

Fig 2-5 Subclasses of the Writer class

No conversion occurs when characters are written to, or read from, a byte stream. Characters are converted from Unicode to the local machine representation of characters when a character stream is written.

2.6 Threads

In Java, working with threads can be easy and productive. In fact, threads provide the only way to effectively handle a number of tasks. Threads are integral to the way Java works.

2.6.1 Introducing Threads

Conceptually, a thread is a flow of control within a program. A thread is similar to the more familiar notion of a process, except that multiple threads within the same application share much of the same state--in particular, they run in the same address space. Multiple threads in an application have the some problems, you can't have several threads trying to access the same variables without some kind of coordination. A thread can reserve the right to use an object until it's finished with its task.

2.6.2 The Thread Class and the Runnable Interface

A new thread is born when we create an instance of the java.lang.Thread class. The Thread object represents a real thread in the Java interpreter. With it, we can start the thread, stop the thread, or suspend it temporarily. The constructor for the Thread class accepts information about where the thread should begin its execution. We use the Runnable interface to create an object that contains a "runnable" method by implementing the java.lang.Runnable interface. Every thread begins its life by executing a run( ) method in a particular object. It is public, takes no arguments, has no return value, and is not allowed to throw any exceptions.

2.6.3 Creating and starting threads

A newly born Thread calls its start( ) method. The thread then wakes up and proceeds to execute the run( ) method of its target object. start( ) can be called only once in the lifetime of a Thread. Once a thread starts, it continues running until the target object's run( ) method completes, or we call the thread's stop() method to kill the thread permanently.

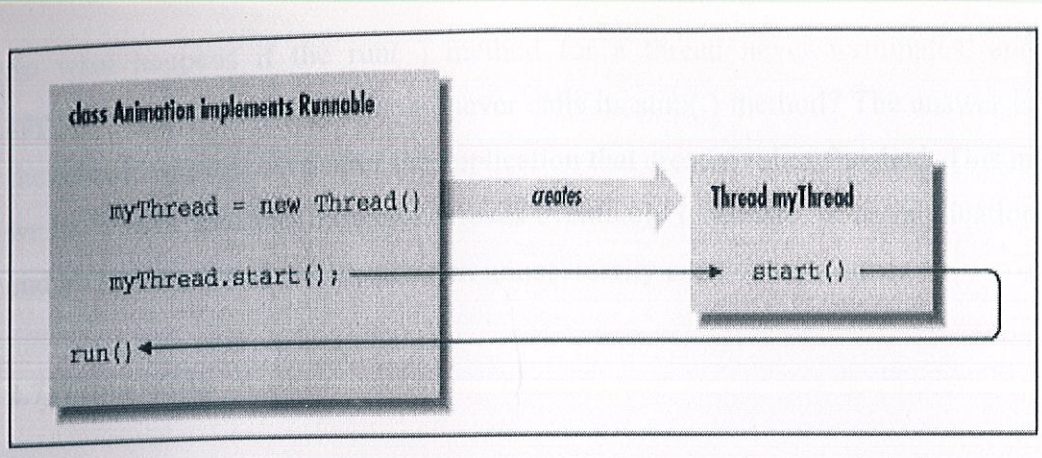

Fig.2-6 Depicts a Runnable object that creates and starts its own Thread.

2.6.4 Controlling Threads

We have seen the start ( ) method used to bring a newly created Thread to life. Three other methods let us control a Thread's execution: stop( ), suspend( ), and resume( ). None of these methods take any arguments; they all operate on the current thread object. The stop( ) method complements start( ); it destroys the thread. start( ) and stop( ) can be called only once in the life of a Thread. By contrast, the suspend( ) and resume( ) methods can be used to arbitrarily pause and then restart the execution of a Thread.

Another common need is to put a thread to sleep for some period of time. Thread.sleep( ) is a static method of the Thread class that causes the currently executing thread to delay for a specified number of milliseconds:

Thread.sleep( ) throws an InterruptedException if it is interrupted by another Thread.[1] When a thread is asleep, or otherwise blocked on input of some kind, it doesn't consume CPU time or compete with other threads for processing.

[1] The Thread class contains an interrupt( ) method to allow one thread to interrupt another thread.

2.6.5 A Thread's Life

A Thread continues to execute until one of the following things happens:

- It returns from its target run( ) method
- It's interrupted by an uncaught exception
- Its stop( ) method is called

So what happens if the run( ) method for a thread never terminates, and the application that started the thread never calls its stop( ) method? The answer is that the thread lives on, even after the application that created it has finished. This means we have to be aware of how our threads eventually terminate, or an application can end up leaving orphaned threads that unnecessarily consume resources.

2.7 Networking

One of Java's great strengths is painless networking. The programming model you use is that of a file; in fact, you actually wrap the network connection (a "socket") with stream objects, so you end up using the same method calls as you do with all other streams. Java's multithreading is handy when handling multiple connections at once.

2.7.1 Identifying a Machine

Of course, in order to tell one machine from another and to make sure that you are connected with the machine you want, there must be some way of uniquely identifying machines on a network. This is accomplished with the IP (Internet Protocol) address that can exist in two forms:

1. The familiar DNS (Domain Name Service) form. If my domain name is **abc.com,** and suppose I have a computer called **xyz** in my domain. Its domain name would be **xyz.abc.com.** This is exactly the kind of name that you use when you send email to people, and is often incorporated into a World-Wide-Web address.

2. Alternatively, you can use the dotted quad form, which is four numbers separated by dots, such as 123.255.28.120. In both cases, the IP address is represented internally as a 32-bit number (so each of the quad numbers cannot exceed 255), and you can get a special Java object to represent this number from either of the forms above by using the static InetAddress.getByName( ) method that's in java.net. The result is an object of type InetAddress that you can use to build a "socket" as you will see later.

2.7.2 Servers and clients

The whole point of a network is to allow two machines to connect and talk to each other. Once the two machines have found each other they can have a nice, two-way conversation. The machine that is being seeked is called the server, and the one that seeks is called the client. This distinction is important only while the client is trying to connect to the server. So the job of the server is to listen for a connection, and that's performed by the special server object that you create. The job of the client is to try to make a connection to a server, and this is performed by the special client object you create. Once the connection is made, you'll see that at both server and client ends, the connection is just magically turned into an I0 stream object, and from then on you can treat the connection as if you were reading from and writing to a file. This is one of the nice features of Java networking.

2.7.3 Port: A unique place within the machine

An IP address isn't enough to identify a unique server, since many servers can exist on one machine. Each IP machine also contains ***ports***, and when you're setting up a client or a server you must choose a port where both client and server agree to connect. The client program knows how to connect to the machine via its IP address, but ports are responsible for connecting it to the desired service (potentially one of many on that machine). The idea is that if you ask for a particular port, you're requesting the service that's associated with the port number. Typically, each service is associated with a unique port number on a given server machine. The system services reserve the use of ports 1 through 1024, so you shouldn't use those or any other port that you know to be in use. The ***socket*** is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into a socket. In Java, you create a socket to make the connection to the other machine, then you get an Inputstream and Outputstream from the socket in order to be able to treat the connection as an I0 stream object. There are two stream-based socket classes: a Serversocket that a server uses to "listen" for incoming connections and a Socket that a client uses in order to initiate a connection. Once a client makes a socket connection, the **ServerSocket**

returns (via the **accept( )** method) a corresponding server side **Socket** through which direct communications will take place. At this point, you use the methods **getInputStream( )** and **getoutputstream( )** to produce the corresponding **InputStream** and **OutputStream** objects from each **Socket.** These must be wrapped inside buffers and formatting classes just like any other stream object. **ServerSocket** seems to be a bit misnamed, since its job isn't really to be a socket but instead to make a **Socket** object when someone else connects to it When you create a **ServerSocket,** you give it only a port number. You don't have to give it an IP address because it's already on the machine it represents. When you create a **Socket,** however, you must give both the IP address and the port number where you're trying to connect. (On the other hand, the **Socket** that comes back from **ServerSocket.accept( )** already contains all this information.).

2.7.4 Creating Clients and Servers

Using BufferedReader and PrintWriter objects, you can communicate over the network with sockets-all you need is a Domain Name Service (DNS) address for the server and a free port on that server.

In the following case a client program connects to a server program, sends a message, and gets a message back. I will use the DNS 127.0.0.1 (the local host) and use an arbitrary port number 8765. The port number selected should not be in use already and the server and client should connect to the same port.


```
// CLIENT PROGRAM

import java.net.*;
import java.io.*;
class client {
        public static void main(String args[]) throws Exception {
                int character;
                Socket socket = new Socket("127.0.0.1", 8765);
                InputStream in = socket.getInputStream();
                OutputStream out = socket.getOutputStream();
```

```
            String string = "Hello!\n";
            byte buffer[] = string.getBytes();
            out.write(buffer);
            while ((character = in.read()) != -1) {
            System.out.print((char) character);
        }
        socket.close();
        }
}

//SERVER PROGRAM

import java.io.*;
import java.net.*;
public class server {
        public static void main(String[] args ) {
                try {
                        ServerSocket socket = new ServerSocket(8765);
                        Socket insocket = socket.accept( );
                        BufferedReader in = new BufferedReader (new
                        InputStreamReader(insocket.getInputStream()));
                        PrintWriter out = new PrintWriter
                (insocket.getOutputStream(),
                                true);
                        String instring = in.readLine();
                        out.println("The server got this: " + instring);
                        Insocket.close();
                }
                catch (Exception e) {}
        }
}
```

2.8 AWT

AWT (the Abstract Window Toolkit) is the part of Java designed for creating user interfaces and painting graphics and images. It is a set of classes intended to provide everything a developer requires in order to create a graphical interface for any Java applet or application. Most AWT components are derived from the java.awt.Component class as figure below illustrates. (Note that AWT menu bars and menu bar items do not fit within the Component hierarchy.)

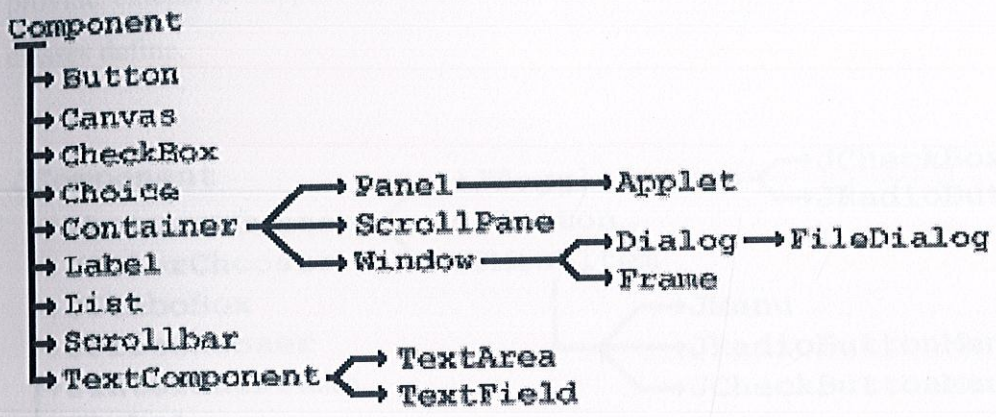

Fig.2-7 Partial Component Hierarchy

The Java Foundation Classes consist of five major parts: AWT, Swing, Accessibility, Java 2D, and Drag and Drop. Java 2D has become an integral part of AWT, Swing is built on top of AWT, and Accessibility support is built into Swing.

The five parts of JFC are certainly not mutually exclusive, and Swing is expected to merge more deeply with AWT in future versions of Java. The Drag and Drop API was far from mature at the time of this writing but we expect this technology to integrate further with Swing and AWT in the near future. Thus, AWT is at the core of JFC, which in turn makes it one of the most important libraries in Java 2.

2.9 SWINGS

Swing is a large set of components ranging from the very simple, such as labels, to the very complex, such as tables, trees, and styled text documents. Almost all Swing components are derived from a single parent called JComponent which extends the AWT Container class. Thus, Swing is best described as a layer on top of AWT rather

than a replacement for it. Figure below shows a partial JComponent hierarchy. If you compare this with the AWT Component heirarchy of previous figure you will notice that for each AWT component there is a Swing equivalent with prefix "J". The only exception to this is the AWT Canvas class, for which JComponent, JLabel, or JPanel can be used as a replacement. You will also notice many Swing classes with no AWT counterparts.

Figure below represents only a small fraction of the Swing library, but this fraction consists of the classes you will be dealing with most. The rest of Swing exists to provide extensive support and customization capabilities for the components these classes define.

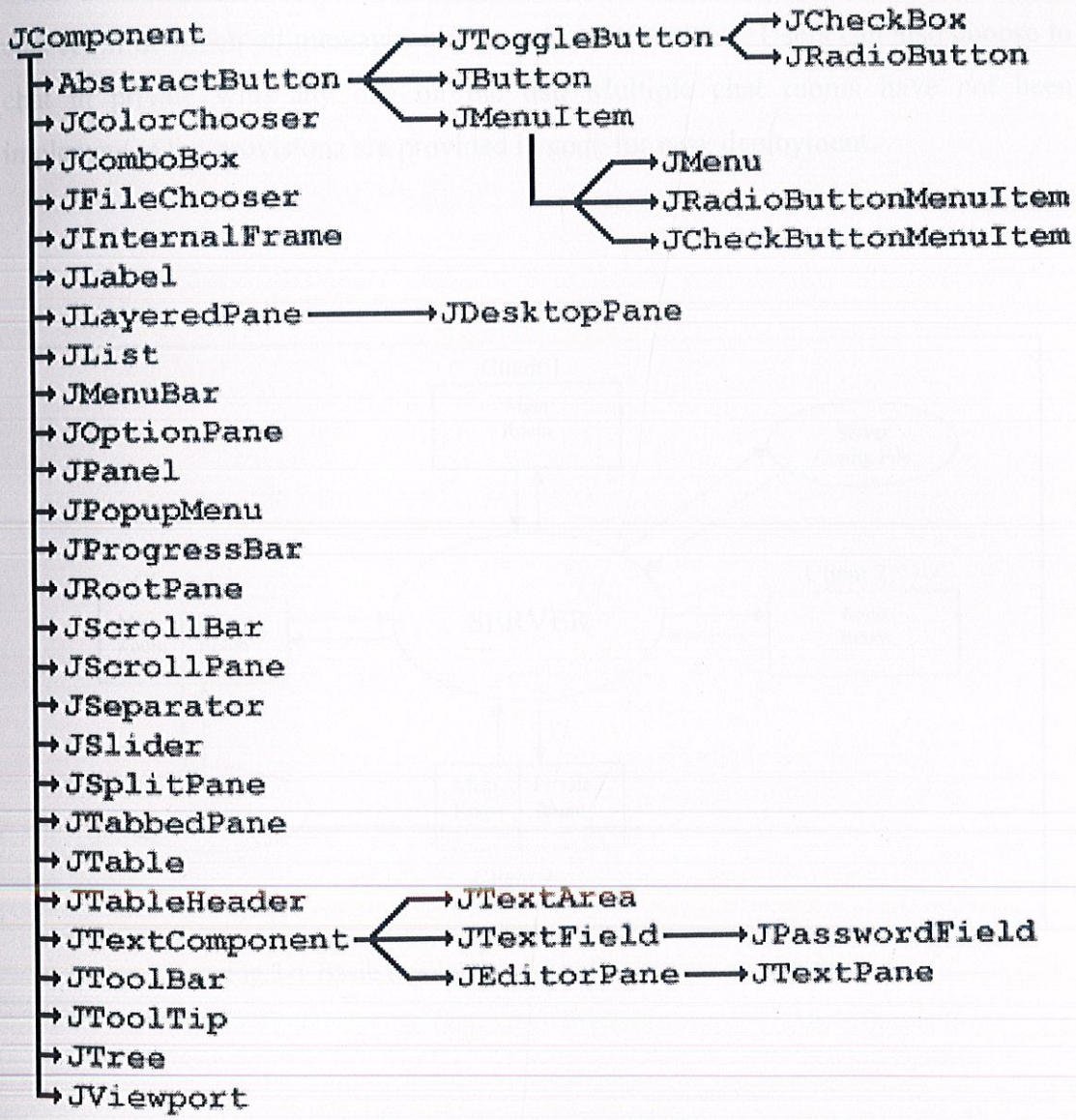

Fig.2-8 Partial Jcomponent Hierarchy

CHAPTER 3

PROPOSED STUDY AND ITS IMPLEMENTATION

3.1 Theory

3.1.1 Outline

MAV is chatting software based loosely on IRC system. There is a central server handling all communications to and from clients. Each user can run the client program and connect to server to start chatting. All clients and server will have list of online users. List is updated as soon as the status of some client changes. There is one main chat room in which all messages can be seen by all clients. Users can also choose to chat in private with any one on the list. Multiple chat rooms have not been implemented but provisions are provided in code for easy deployment.

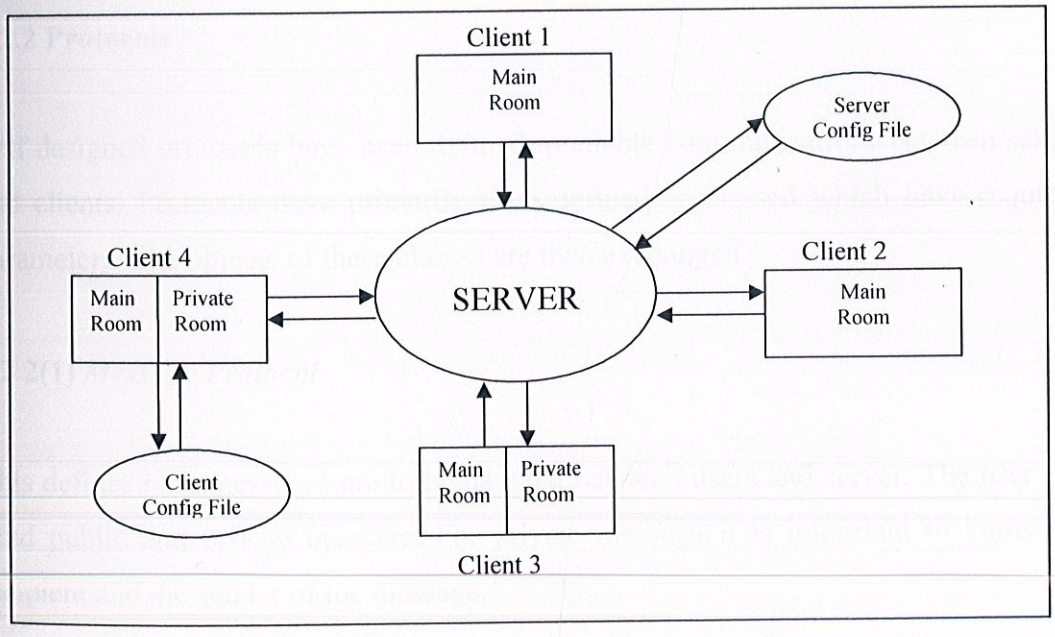

Fig.3-1 Basic topology for the client\server architecture

3.2 Implementation Details

Language Used	Java
Libraries Used	JDK 1.5
IDE	JCreator
Platform	Java runtime environment 5.0
User Interface	Graphical

3.2.1 Client – Server Communication

The server is bound to a fixed socket and listens for connection requests from clients. The clients try to connect to server on this port and predefined host. Once the communication channels are set up, both talk in terms of objects defined as protocols. Upon receiving these objects the program then extracts relevant information and takes appropriate actions. All communications are through server and may change the protocol parameters if required.

3.2.2 Protocols

Self designed protocols have been defined to enable communications between server and clients. Protocols have primarily been defined as classed which have required parameters. The objects of these classes are then exchanged

3.2.2(1) *Message Protocol*

This defines how messages are to be handled between users and server. The user can send public and private message. For private message it is important to know the recipient and the sender of the message.

Fields:

- Audience – public or private message
- RoomNumber – Currently of no use. In future can be used for multiple rooms
- RecieverId – Id of the recipient. Useful only for private messages

- SenderId – Id of sender. Useful only for private messages
- Message – Text that the user wants to send

3.2.2(2) *Client Information Protocol*

This is meant to exchange client information between user and server. When a new client connects to the server its relevant information is kept in an object of this class. Other users are notified of arrival of new client using information from this protocol.

Fields:

- ClientId – Identification number of client within the server.
- ClientName – The login name provided by the user.

3.2.2(3) *Chat Request Protocol*

This protocol is used to notify a client that another client wants to start a private chat with it. A message of this type must be sent before any private conversation can start. This message is sent when user chooses to start a private conversation. Upon receiving this request the recipient's client takes steps to receive private messages from the server by the specified sender.

Fields:

- SenderId – The clientId of the client machine that initiated the request.
- RecieverId – The clientId of the client machine that is to be notified.

3.2.2(4) *Update Client List Protocol*

When a new user logs in to the server all clients have to be notified of this arrival. Also when a user logs out, all users must be notified. This protocol is used to simplify this process. A message of this type with the new clients name is broadcast to all client machines.

Fields:

- Request Type – Indicates if the user has to be added or removed

- ClientName – The name of the client that the information is about

3.2.2(5) *Log Out Protocol*

When a user chooses to logout of the system the server and all other users must be notified. Upon users choice the local client machine sends a message of this type to the server. Upon receiving this message the server forwards it to all clients. Then breaks connection with the client.

There are no fields

3.2.2(6) *Shut Down Protocol*

If the server has to be shut down it must notify the clients. This message is broadcast to all clients that they must close their connections.

There are no fields

3.2.2(7) *Join Chat Room Protocol*

This protocol is reserved for when multiple chat rooms will be implemented.

Fields

- Request Type -- Indicates if the user has to be added or removed
- Room Number – RoomId of the room that the user wants to join

3.2.2(8) *Kicked Out Notice Protocol*

If the administrator chooses to kick out a user the server must send this message to the kicked out client. The client is told out of which room the user has been kicked out of. If the user is kicked out of the main room it is equivalent of a forced log out.

Fields

- RoomNumber – Indicates which room the user has been kicked out of

3.2.2(9) *Connection Notice Protocol*

If the server rejects the connection then this object is sent to the client. The reason might be over occupied server or clients nick already in use.

Fields

- Status – indicates whether the connection was accepted or rejected by the server.

3.2.3 User Interfaces:

3.2.3(1) *Server Interface*

The interface has been developed in Swing. Interface has been kept separate from the network processes. The main components of the server interface are as follows

- Messages Area: Connection acceptance, rejection, login messages are shown here
- List of Online Users: On the right side of the message window is the list of users that are connected to the server currently. A user can be selected from this list by clicking on name.
- Configure Server Dialog: This dialog is shown when option is selected from the menu. This dialog will allow new values and saving to configuration file.
- Main Menu: The options available for the server. The options include configure server, shutdown server.

3.2.3(2) *Client Interface*

The interface has been developed in Swing. Interface has been kept separate from the network processes. The main components of the client interface are as follows

- Message Tabs: These are the conversation tabs. All conversation windows are kept within these tabs.
- Message Entry Field: This is place at the bottom of the window. This is where the user enters whatever message he/she wants to send. Message is sent by either pressing enter or pressing the send button. Where the message is sent depends on

which tab is open

- Online User List: This list shows all the users who are logged in at the server. Double clicking on a user will open a conversation window with him.
- Configure Dialog: This dialog is shown when option is selected from the menu. This dialog will allow new values and saving to configuration file. You can change server host name and port.
- Main Menu: The options available for the server. The options include connect, disconnect, configure, exit, close current tab, close all tabs, Help

3.3 Snapshots

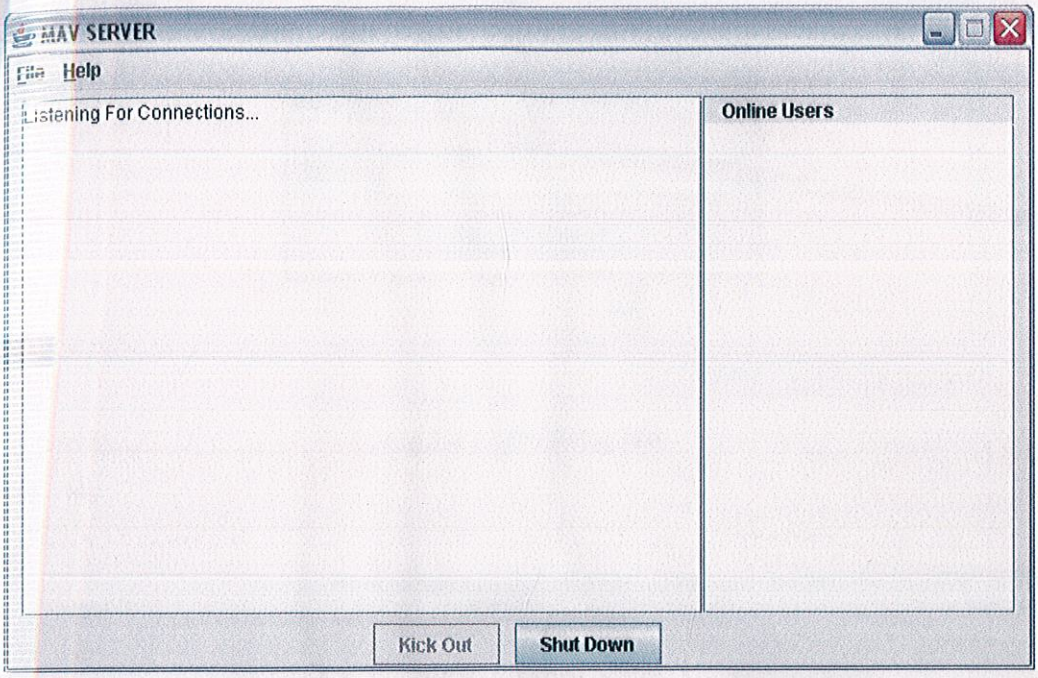

Fig.3-2 The Server window in default mode

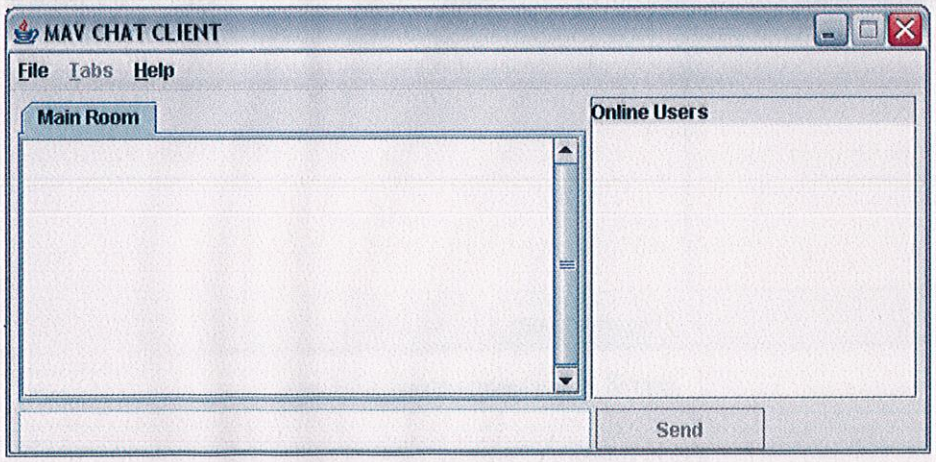

Fig.3-3 The Client window in default mode

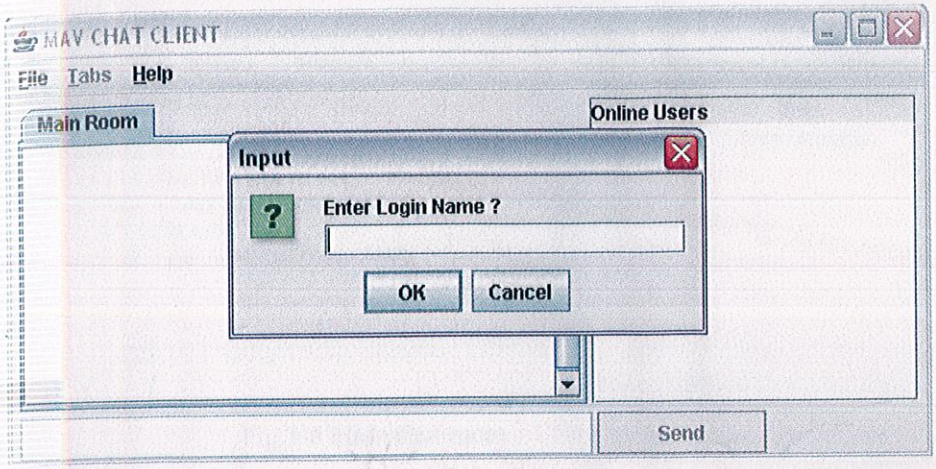

Fig.3-4 Login dialog box

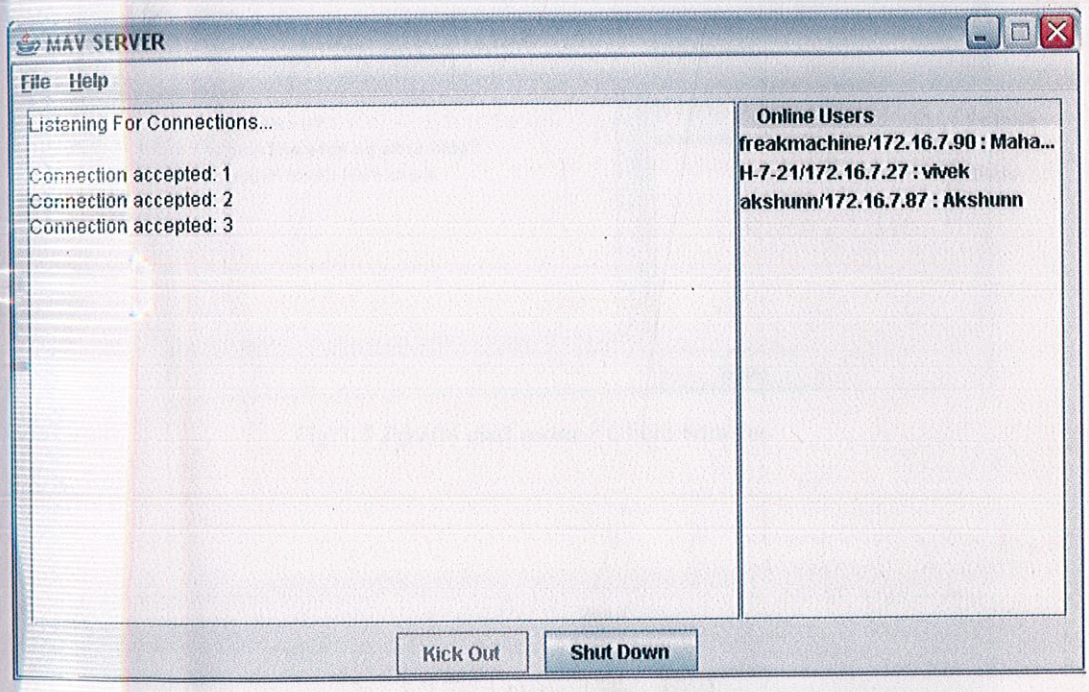

Fig.3-5 Clients connected to Server

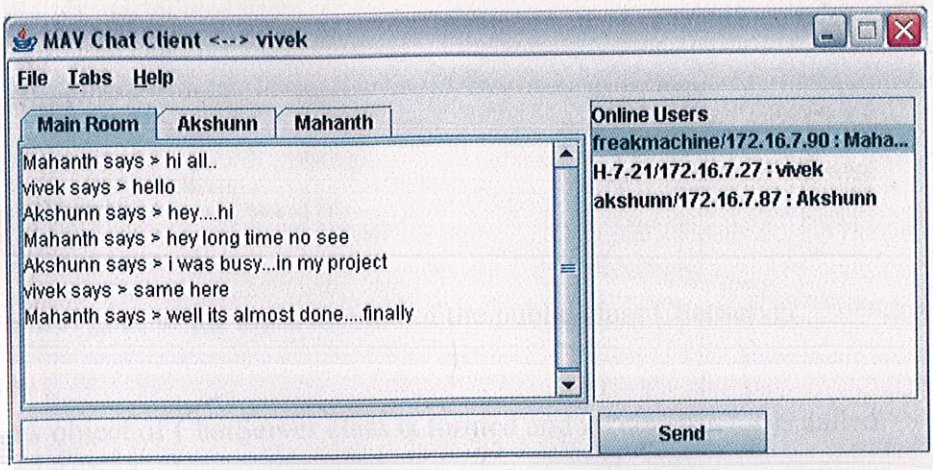

Fig.3-6 Main chat room – Client window

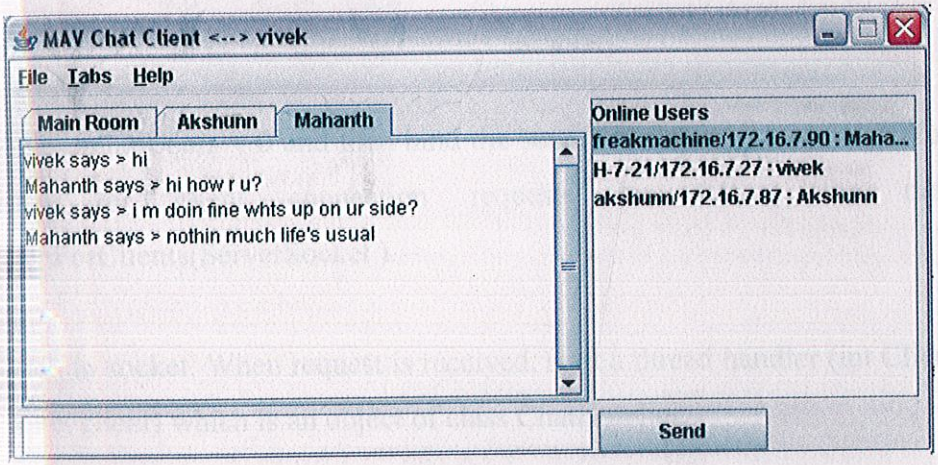

Fig.3-7 Private chat room – Client window

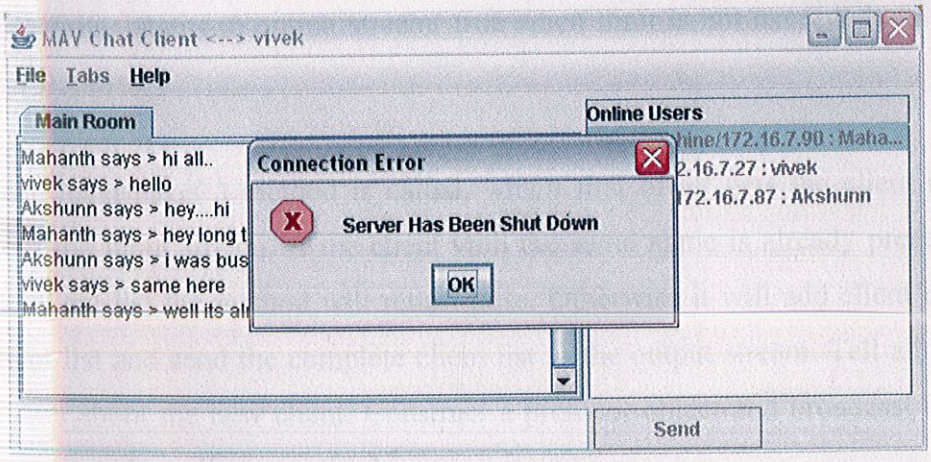

Fig.3-8 Server shutdown

3.4 Algorithm

3.4.1 Chat Server

1. Control goes to the main method of the public class ChatServer.

2. New object of ChatServer class is formed and its constructor is called.

3. The constructor calls the method getConfiguration( ), which configures serverPort number and serverLimit number (By default 1665 and 20 respectively).

4. Make onlineUsers = 0 and then bind the server on socket, show interface and listen for client connection requests by calling the method listenForClients(ServerSocket).

5. Listen to socket. When request is received, start a thread handler (int ClientID, Socket client) which is an object of class ChatHandler.

6. Now start method is called by handler. This method creates the ouput stream and input stream and checks if client limit is exceeded or not and write appropriate status to output stream(true when limit is not exceeded and vice versa).

7. Then handshake( ) method is called, which first of all gets the client name from the input stream. If the client with the same name is already present in the client list the method will return false. Otherwise it will add client to the client list and send the complete client list to the output stream. Tell all other clients about the new client. Construct a protocol object and broadcast it and increment the online users by 1.

8. If handshake returns false then return to the method listenForClients( ). Otherwise create a new thread object and call the object's start method which will execute the code written in the run method (an overridden method).

9. Now the run( ) method continuously listens to input stream for messages from "this" client.

   If message received

   - Public message then broadcast
   - Private message then locate the intended receiver. Get the id of recipient (recieverId), the reference of handler and send message on its output stream.

   If chat request is received

   - Tell the intended recipient that this client wants to start a private chat. Locate the intended receiver. Get the id of recipient (recieverId), the reference of handler and send request on its output stream.

   If Logout request is received

   - Then decrement the online users by one and break from the loop after which the stop() method is executed

3.4.2 Chat client

1. Control goes to the main method of the public class ChatClient.

2. New object of ChatClient class is formed and its constructor is called.

3. The constructor sets the Boolean variable connected = false and creates an object listener of class InputListener.

4. When the client enters a login name via a graphical interface to connect to a server the control transfers to the method connectToServer( ).

5. The method connectToServer( ) calls the method getConfiguration( ), which configures serverSocketNumber and serverAddress (By default 1665 and localhost respectively).

6. An object addr of the inbuilt java class InetAddress is created and assigned server address. Now the object of class Socket, socket connects to the server by calling the constructor using addr and serverSocketNumber as an argument.

7. Now it creates the input stream and the output stream and the control shifts to the method handShake( ).

8. The handShake( ) method reads the status from the input stream and if it is true then writes the login name of the client on the output stream and if it is false displays the message "Name Already In Use. Change Login Name" and returns false. Get the client list from the input stream and assigns to client id the value of number of clients in the clientList minus 1 and returns true.

9. Now control returns back to the method connectToServer( ) and if handshake() returns false then connection to the server is not established, otherwise if it is true make Boolean variable running = true, which is a field in the InputListener class. If connecting for the first time, start the listener object

(which will execute the method run( )), make Boolean variable connected = true and the connection to the server is established.

10. Now the run( ) method continuously listens to input stream for messages from the server and stores it in the object server message of class Object.

    If message received is from another client connected to server then

    - If message is public show in main room tab.
    - If message is private show in private room tab.

    If chat request is received

    - It will accept the request and open a new tab corresponding to that senderId.

    If UpdateList request is received

    - Then update the clientList according to the value of the Boolean variable requestType. If it is true add client in the clientList else remove the client.

    If ServerShutDown request is received

    - Then disconnect from the server and display the same.

    If KickedOutNotice request is received

    - Then disconnect from the server and display the message "Server Kicked You Out".

11. If user logs out then the control goes to the method disconnectFromServer(Boolean reason) and sets the Boolean variable running = false and if reason = true then write the object of class logout to the output stream. Close the output stream and the socket, clear the client list and make the Boolean variable connected = false.

+++

CHAPTER 4
CONCLUSION

Study of Java Programming and Network Programming through Java has been done. A Centralized Platform Independent Instant Messenger Application using the same was developed. Both the server and client are reconfigurable. The Server reserves the right to decide the maximum number of clients and disconnect any particular client during a session. Private rooms for communication between any pair of clients have been provided along with a main room for conferencing by all clients connected.

4.1 Possible Future Enhancements

The following features can be incorporated in future to enhance its functionality:

- File Transfer – File Transfer between clients.
- VoIP –Voice over Internet Protocol – Voice Chat Enhancement.
- Chat History Logging - A Log file containing the recent chat history.
- Compression of Data - For Faster and Secure Delivery over large networks.
- Multiple chat rooms – Multiple Chat Server Connection capability in the client.

Appendix

Appendix A:

Main Classes and their functionalities.

1. Chat Server:

   This is the main class of the MAV Chat Server. It provides the core functionality of the server and is responsible for handling clients and their connections. Information about all clients is kept in this class. Error handling has been done in this class. An object of chat handler is created fro each client.

2. Chat Handler:

   This is an inner class in the chat server. This is basically responsible for handling each individual clients i.e. their sockets, the input and output stream. It runs as a separate thread for each client. Creation and deletion of the connection and broadcasting of message is done in this class.

3. Server Interface:

   This is the class responsible for the GUI provided to the server administrator. The administrator is provided with an easy to use interface providing multiple paths to carry out a particular task. Menus have been incorporated and buttons are also placed on the screen to provide assistance to the user of the server. Error Messages from the chat server class are displayed in this class.

4. Chat Client:

   This is the main class of the MAV Chat Client. It provides the core functionality of the client and is responsible for making connection with the server and sending and receiving messages and classifying them according to the used protocols. Error handling has been done in this class.

5. Client Interface:

   This class is responsible for the GUI of the client part of the application. Again a user friendly interface has been provided to the user for easy usage of the software. Multiple paths have been provided to carry out a particular task. User can view the messages sent in the main room as well switch to the different tabs provided to use the private chat option of the application. Errors captured in the chat client class are displayed in this class.

6. Input Listener:

   This is an inner class in the chat client. This class is sub class of the thread class. This class listens to the incoming transmissions from the chat server. The objects are then classified according to the protocols and then the appropriate action is taken by the other classes.

Appendix B: Error Messages

Server Error Messages

Message	*Reason*	*Solution*
Class of a serialized object cannot be found	Error in communication between server and client	Press OK. If keeps repeating shutdown
Something is wrong with a class used by serialization.	Error in communication between server and client	Press OK. If keeps repeating shutdown
Control information in the stream is inconsistent.	Error in communication between server and client	Press OK. If keeps repeating shutdown
Primitive data was found in the stream instead of objects.	Error in communication between server and client	Press OK. If keeps repeating shutdown
Cannot Setup Connection	Error establishing connection with client	Just Press OK. If keeps repeating shutdown
Cannot Save Configuration File	Error saving configuration options to file	Retry Configuring. If keeps repeating check if file is corrupted
Configuration File Not Found, Using Defaults	Error finding and opening configuration file.	Press OK. If keeps repeating check if file has been deleted.
Error Reading Configuration File, Using Defaults	File found but cannot be read. File may be created	Retry Configuring. If keeps repeating check if file is corrupted
Cannot Start Server	Another program may be using the port on the machine	Shutdown other program if possible. Else change port number in the configuration file

Message	*Reason*	*Solution*
Error closing connection to client	Error while trying to break connection with client	Press OK.

Client Error Messages

Message	*Reason*	*Solution*
Server Has Been Shut Down	Server Has Been Shut Down	Press OK. Reconnect later
Server Kicked You Out	Administrator kicked you out	Press OK. Reconnect later
Class of a serialized object cannot be found	Error in communication between server and client	Press OK. Reconnect later
Something is wrong with a class used by serialization.	Error in communication between server and client	Press OK. Reconnect later
Control information in the stream is inconsistent.	Error in communication between server and client	Press OK. Reconnect later
Primitive data was found in the stream instead of objects.	Error in communication between server and client	Press OK. Reconnect later
Cannot Save Configuration File	Error saving configuration options to file	Retry Configuring. If keeps repeating check if file is corrupted
Configuration File Not Found, Using Defaults	Error finding and opening configuration file.	Press OK. If keeps repeating check if file has been deleted.
Error Reading Configuration File, Using Defaults	File found but cannot be read. File may be created	Retry Configuring. If keeps repeating check if file is corrupted
Host Not Found, Reconfigure	Server host machine cannot be found.	Check your configuration, change if neccessary

Message	*Reason*	*Solution*
Server Not Found, Check If Server Exists	No server could be found listening to the port on the specified machine	Server may not have been started. Retry later.
Cannot Create Data Stream, Closing Client	Connection Established but data stream cannot be resolved	Reconnect later
Name Already In Use. Change Login Name	Someone has already logged in with your chosen name	Reconnect and choose a different name.
Maximum User Limit Reached. Server Rejected Connection	Maximum numbers of users allowed have connected to server.	Try connecting later

Bibliography

Study of Java Programming and Network Programming through Java

1. Thinking in Java, 2nd Edition – BruceEckel – http://www.BruceEckel.com – Java Programming.

2. Java Network Programming, 2nd Edition – Merlin Hughes, Michael Shoffner, Derek Hamner – http://nitric.com/jnp/ -- Network Specific Java Programming.

3. Java 2 Black Book – Steven Holzner – Java Programming and Swing.

4. Sun's Java Tutorial -- http://java.sun.com/docs/books/tutorial – Swing and Updated Methods.

5. Java 2 Swing – O'Reillys.

6. Beginning Java™ 2, JDK™ 5 Edition – Ivor Horton – Java Programming.

7. www.wikipedia.com.